ILP-Based Energy Minimization Techniques for Banked Memories

OZCAN OZTURK
Bilkent University
and
MAHMUT KANDEMIR
The Pennsylvania State University

Main memories can consume a significant portion of overall energy in many data-intensive embedded applications. One way of reducing this energy consumption is banking, that is, dividing available memory space into multiple banks and placing unused (idle) memory banks into low-power operating modes. Prior work investigated code-restructuring- and data-layout-reorganization-based approaches for increasing the energy benefits that could be obtained from a banked memory architecture. This article explores different techniques that can potentially coexist within the same optimization framework for maximizing benefits of low-power operating modes. These techniques include employing nonuniform bank sizes, data migration, data compression, and data replication. By using these techniques, we try to increase the chances for utilizing low-power operating modes in a more effective manner, and achieve further energy savings over what could be achieved by exploiting low-power modes alone. Specifically, nonuniform banking tries to match bank sizes with application-data access patterns. The goal of data migration is to cluster data with similar access patterns in the same set of banks. Data compression reduces the size of the data used by an application, and thus helps reduce the number of memory banks occupied by data. Finally, data replication increases bank idleness by duplicating select read-only data blocks across banks. We formulate each of these techniques as an ILP (integer linear programming) problem, and solve them using a commercial solver. Our experimental analysis using several benchmarks indicates that all the techniques presented in this framework are successful in reducing memory energy consumption. Based on our experience with these techniques, we recommend to compiler writers for banked memories to consider data compression, replication, and migration.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers; memory management; optimization*

This article extends the material presented in *Proceedings of the ACM Great Lakes Symposium on VLSI* (*GLSVLSI*) [Ozturk and Kandemir 2005a] and *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE)* [Ozturk and Kandemir 2005b]
This work is supported in part by NSF Career Award 0093082 and by a grant from GSRC.
Authors' addresses: O. Ozturk, Computer Engineering Department, Bilkent University, 06800 Ankara, Turkey; email: ozturk@cs.bilkent.edu.tr; M. Kandemir, Penn State University, 354C Information Sciences and Technology Building, University Park, PA 16802; email: kandemir@cse.psu.edu.

General Terms: Experimentation, Management, Design, Performance

Additional Key Words and Phrases: Memory banking, compilers, low-power operating modes, data compression, migration, replication, DRAM

ACM Reference Format:

Ozturk, O. and Kandemir, M. 2008. ILP-based energy minimization techniques for banked memories. ACM Trans. Des. Autom. Electron. Syst. 13, 3, Article 50 (July 2008), 40 pages, DOI = 10.1145/1367045.1367059 http://doi.acm.org/10.1145/1367045.1367059

1. INTRODUCTION

Main memories can consume a significant portion of overall energy in embedded applications [Catthoor et al. 1998; Farkas et al. 2000]. This is particularly true for the class of embedded systems that execute data-intensive applications such as multimedia processing, which manipulates multidimensional arrays of signals using a series of nested loops. Numerous techniques have been proposed in the past for reducing memory energy consumption, including circuit-level techniques and high-level approaches. While circuit-level techniques [Azizi et al. 2003; Moon et al. 2002] targeting energy efficiency are extremely important, high-level approaches [Cao et al. 2002; Sudarsanam and Malik 2000; Saghir et al. 1996; Panda 1999] at the architectural and software levels can also play a major role in shaping the overall memory energy behavior and optimizing it.

Recently, *banking* has been used as a popular method for reducing memory energy consumption. In banking, memory space is divided into multiple banks, each of which can be controlled independently of the others. Experiments performed by several research groups [Fan et al. 2002; Delaluz et al. 2003, 2001; Lebeck et al. 2000; Farrahi et al. 1998] reported significant reductions in memory energy due to banking. Memory banking can be useful from the energy consumption perspective in two ways. First, each access in a banked architecture is to a single bank, and consequently experiences a smaller load capacitance as compared to an access to a large monolithic (unbanked) memory architecture. Therefore, even if no special optimization is incorporated, using a banked memory itself reduces memory energy consumption. Second, when available, low-power operating modes and accompanying code- and data transformations can be used to further increase memory energy-savings. Prior research considered both hardware-based [Delaluz et al. 2001] and software-based [Lebeck et al. 2000; Delaluz et al. 2003] low-power operating-mode management schemes.

Focusing on a banked DRAM-based architecture, this article studies several techniques for reducing memory energy in a banked system beyond what could be achieved through low-power modes alone. In this work, we try to answer the following questions.

—How does the data assignment to memory banks affect the energy consumption?

—Is it sufficient to have uniform memory banks for maximum energy-savings, that is, could nonuniform banking bring any benefits over uniform banking?

—Can data migration reduce energy consumption further?

—How much additional energy savings can data compression bring?
—How can we use data replication to reduce energy consumption further, without excessively increasing memory-space requirements?

Our main goal in this article is to present ILP (integer linear programming) formulations for these problems, and to quantify their impact in saving memory energy using a set of applications. One of the common characteristics/assumptions of most prior memory-banking-related studies is that they assume all banks to be of the same size, that is, to be uniform. This prevents several energy reduction opportunities by unnecessarily restricting the data mapping. In particular, in embedded systems that execute a single application, one can conceive of a customized banking strategy where the different banks can be of different sizes. Such a banking is referred to as nonuniform banking in this work. The first contribution of this article is to present an ILP-based approach that decides the best (nonuniform) bank architecture and accompanying data mapping for a given array-based embedded application. We show through our experiments that working with customized nonuniform memory banks brings significant energy savings over an alternate strategy that works with optimal data mapping but under uniform bank size (which is also formulated in this article using ILP).

While our empirical evaluation of the nonuniform banking shows promising results, one can achieve further energy savings by not fixing the location of each data unit (i.e., its bank) for the entire execution, that is, by migrating data across banks during the course of program execution. The second contribution of this article is an ILP formulation of the data migration problem in the context of a nonuniform bank architecture. Our experimental evaluation of data migration shows that it brings additional energy savings over nonuniform banking, ranging from 24.3% to 33.9%. This is made possible because migration can place data blocks with similar access patterns/lifetimes into the same set of banks, thereby increasing the chances for better utilizing low-power modes.

Similarly, data compression squeezes data blocks in memory, which in turn allows a better use of available DRAM space and allows to increase the number of inactive (idle) banks, which are candidates for being put in low-power operating modes. Data compression can increase energy savings further by cooperating with migration. However, now, whenever the execution accesses the compressed data, it needs to be decompressed. Consequently, compression must be used with care for select data blocks. Note that our approach is orthogonal to the selection of compression/decompression algorithm employed. Since our approach is compiler based, we have to work with a reasonable high-level representation for code optimization purposes (otherwise, our approach would have to be modified each time one uses a new compression algorithm). In our approach, the costs for compression and decompression are taken into account (i.e., they are input to our formulation). If a particular compression algorithm is chosen, we can change the associated costs accordingly. The rest of our approach does not require any modification, which is critical from the portability viewpoint. The third contribution of this work is

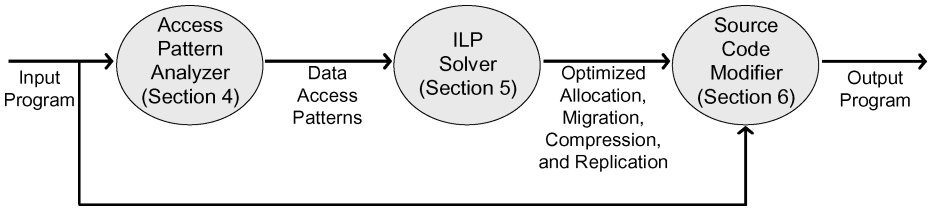

Fig. 1. High-level view of our approach.

an ILP formulation of the data compression problem in a nonuniform bank architecture.

One of the common assumptions implicitly made by prior studies on banking is that each data block has only a single copy in the banked memory system. This assumption, while preferable from the viewpoint of reducing the total memory footprint of program data, may cause unnecessary power consumption in the context of banked memories. We propose a novel data placement strategy that can make use of data replication. However, since any replication increases the overall memory footprint of data, it should be applied with care. Therefore, the problem attacked is to reduce power consumption while keeping the amount of replication under control. In other words, given a fixed amount of extra memory that we can use for replicating read-only data blocks, our approach makes best use of this space for reducing the power consumption of the banked memory by exploiting the available low-power modes. As a fourth contribution, this article proposes an ILP-based solution for the data replication problem in the nonuniform bank architecture. Our experiments show that data replication brings 14% energy savings on average (over the case when we use low-power modes without any replication but with optimal data placement). Also, when used along with the other optimizations in this article, replication further reduces energy consumption by 12% on top of the energy reduction provided by these optimizations.

We formulate each of these problems using ILP and solve them using a commercial solver. Our approach determines the optimal memory-bank sizes, as well as optimal data migration, compression, and replication patterns, based on the data-access pattern information extracted by the compiler. The optimal migration, compression/decompression, and replication strategies determined by the ILP solver are then fed to the compiler, which in turn modifies the application code automatically to insert explicit migration, compression/decompression, and replication calls (instructions). Figure 1 illustrates a high-level view of our approach. Our results show that the proposed techniques can help reduce the overall memory-system energy consumption for the set of array-intensive benchmarks in our experimental suite. The results also reveal the solution times taken by the ILP-based approach to be within tolerable range.

The rest of this article is organized as follows. The next section gives a detailed discussion of related work on banked memories. Section 3 gives a brief description of banked memories and low-power operating modes. Section 4 explains the data-access pattern extraction strategy employed by our compiler.

Section 5 describes our ILP variables, constraints, and overall problem formulation presented to the ILP solver. Section 6 discusses the code modifier through an example. Section 7 describes our experimental platform, benchmarks, and methodology, and presents the results. Section 8 concludes with a summary of the work.

2. RELATED WORK

Prior work has considered power management of banked memories from the hardware-, OS-, and compiler viewpoints. Delaluz et al. [2001] investigate software and hardware techniques to exploit DRAM mode-control capabilities. A compiler-directed mechanism, along with a runtime approach referred to as the self-monitored technique, has been proposed in that work. Lebeck et al. [2000] explore the interaction of page placement with static and dynamic hardware policies to exploit low-power operating modes. A trace-driven and an execution-driven simulator have been used in their study. Fan et al. [2002] study OS-based DRAM power-control policies. Fan et al. [2001] present memory-controller policies for DRAM architectures with low-power operating modes. The impact of classical loop optimizations on energy consumption of banked memories has been evaluated in Kandemir et al. [2002]. In Kandemir et al. [1999], the authors present an iteration-space reordering technique for banked memories. Farrahi et al. [1998] discuss how a sleep mode can be exploited for memory partitions. The impacts of loop optimization (loop splitting and loop distribution) and array placement strategies on a banked off-chip memory architecture are presented in Delaluz et al. [2000]. Sudarsanam and Malik [2000] and Saghir et al. [1996] discuss techniques for exploiting dual banks for ASIPs and DSPs, respectively. Panda [1999] addresses the problem of incorporating the application-specific customization of a memory-bank configuration into behavioral synthesis.

The work described in this article builds upon this prior work and shows that it is possible to further increase the energy savings that come from exploiting the low-power operating modes available in banked memories. Apart from the memory system, energy-saving techniques have also been proposed for CPUs [Jejurikar et al. 2004; Kim et al. 2004; Zhuo and Chakrabarti 2005; eun Lee et al. 2003; Bunda et al. 1995], caches [Ghose and Kamble 1999; Inoue et al. 2002, 1999; Kamble and Ghose 1997; Kim et al. 2001; Su and Despain 1995; Kaxiras et al. 2001; Flautner et al. 2002], disks [Gurumurthi et al. 2003; Douglis et al. 1995; Lebeck et al. 2000; Helmbold et al. 1996; Greenawalt 1994], and other I/O units [Chandra and Vahdat 2002; Kesselman et al. 2005; Choi et al. 2002; Chang et al. 2004; Anand et al. 2003]. These approaches are orthogonal to the memory-oriented techniques presented in this article.

3. PRELIMINARIES ON BANKED DRAM ARCHITECTURE AND LOW-POWER OPERATING MODES

The architectural model assumed in this work is based on a multibank memory system. We assume a banked memory architecture similar to RDRAM [Rambus 1999], in which banks can be placed into low-power modes independently. More specifically, each memory bank can be in one of four operating modes, namely,

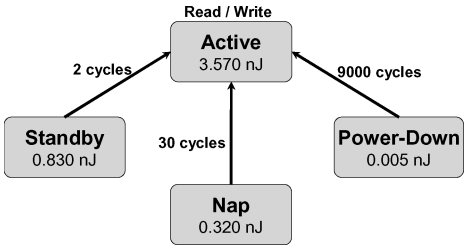

Fig. 2. Energy consumption (per cycle) and resynchronization costs for our operating modes.

active, *standby*, *nap*, or *power-down*, at any point during execution. While we report experimental results with these modes only, our approach can easily be modified to work with any number of low-power modes where available. Read/write requests are serviced only in active mode. The low-power operating modes (i.e., standby, nap, and power-down) can be used only if the bank is not currently servicing a memory request.

Low-power operating modes are typically implemented by disabling certain parts of the DRAM chip and, consequently, each low-power mode typically has a different energy consumption and a different resynchronization cost from the other modes. Resynchronization cost (reactivation penalty) is mainly due to bringing a low-powered memory bank back to active mode. The tradeoff is that decreasing energy consumption using a more aggressive mode results in an increase in cycles. Figure 2 shows energy consumption (per cycle) and resynchronization costs (in cycles) for typical, RDRAM-like low-power operating modes. In this figure, each mode transition is attached a resynchronization cost. As can be seen, when choosing a low-power mode for an idle bank, there is a tradeoff to be accounted for between energy savings and performance penalties. The *bank-interaccess time,* the time between successive accesses to the same memory bank, is the main factor in selecting the most suitable low-power operating mode to use. This is due to the following observation. Frequent transitions between low-power and active modes may cause intolerable performance and energy penalties. Therefore, a more power-saving mode should be selected only if the bank idleness (interaccess time) is predicted to be sufficiently long. Note that while it is also possible in principle to turn off (disable) a memory bank completely, the problem with this approach is that it destroys the contents of the memory bank in question. This is in contrast to the low-power operating modes considered in this work (i.e., standby, nap, and power-down), where the contents of the memory bank are retained while in low-power mode. Another important point we want to mention is in regard to the goal of banking. Banking in the systems we target is for reducing energy consumption. Therefore, it is different from the conventional memory interleaving used in high-performance systems for increasing throughput.

4. BLOCK-ACCESS PATTERN EXTRACTION

Recall from Figure 1 that the first step of our approach is to extract data-access pattern information from the application code. While it is possible to do this

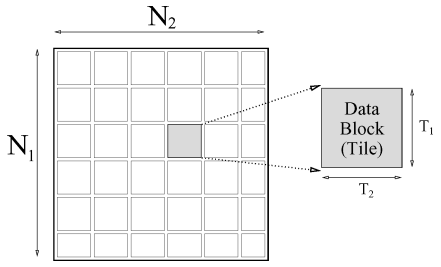

Fig. 3. Dividing a two-dimensional array into data blocks (tiles). Note that all tiles are of the same size, except maybe at the boundaries of the array.

by profiling the code under consideration, the resulting access pattern may be very sensitive to the particular input used in profiling. Instead, in this work, we use static compiler analysis to extract data-access patterns. The following paragraphs discuss the details of our compiler-directed access-pattern analyzer.

Embedded programs constructed using loop nests (with compile time known bounds) and array accesses (with affine subscript expressions) are the main focus in this article. Such codes frequently occur in the embedded image/video-processing domain [Catthoor et al. 1998; Ye et al. 2000]. An optimizing compiler can analyze these loop-intensive applications with regular data-access patterns. Since frequent transitions between low-power modes and active mode can be very costly, an early design decision we made is to perform these mode transitions at well-defined program points. In our implementation, these transition points (which delimit the boundaries of execution phases) are expressed in terms of loop iterations. Specifically, in this article we adopted the concept of a *step* to define these transition points. Although, in theory, we have the flexibility to assign any number of iterations between two transition points, these points should be selected carefully. In other words, in moving from one step to another during execution, the data-access pattern should exhibit significant variation.

The unit of data that is being stored in banks (and also the unit of data migration, compression, and replication across banks) is a *data block* (also referred to as *data tile*). Figure 3 shows a two-dimensional array (X) that is logically divided into data blocks. All the data blocks have the same size, except possibly at the boundaries of the array. An example loop nest that accesses an array X through two references with affine subscript expressions $X[i+2, j-1]$ and $X[i,j]$ is shown on the left side of Figure 4. The blocked (tiled) version of the original loop nest is given on the righthand side of the same figure. In this code, loops i' and j' iterate over the data blocks, and are referred to as the *block* (*intertile*) *iterators*. Loops ii and jj, on the other hand, are called *intratile iterators*, and iterate over the elements of a given data block.

A main factor which affects the data-access pattern is the data-block size. In our experiments, we manually selected suitable data-block sizes for a given application. An optimizing compiler can be used in the future to automate the block-size selection process. Figure 5(a) shows two two-dimensional arrays of the same size divided into data blocks. Now, the code fragment shown in Figure 5(b), written in a pseudolanguage, accesses the data blocks 0, 4, 5, 6, 7,

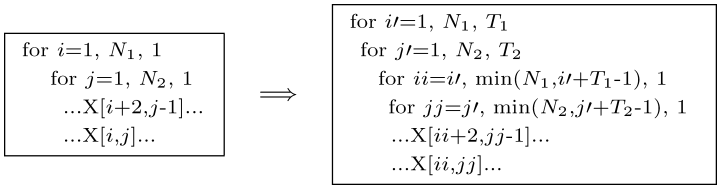

Fig. 4. An example loop nest written in a pseudo high-level language (left) and its blocked (or tiled) version (right). Each data block (tile) is of size $T_1 \times T_2$ array elements, and the transformed loop nest is structured based on this tile size.

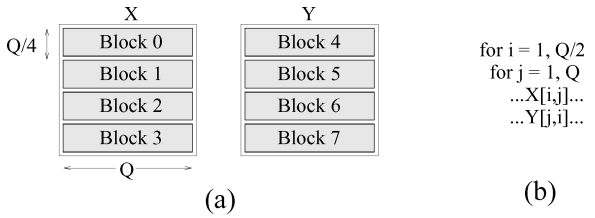

Fig. 5. Two two-dimensional arrays (X and Y) divided into four blocks each; (b) example code fragment operating on these arrays.

and 1, under the data-tile partitioning given in Figure 5(a). Specifically, when this loop nest is executed, the order in which data blocks are accessed is 0, 4, 5, 6, 7, 1. Assuming that the entire code fragment is considered as a single execution step, these are also the blocks accessed in this step. However, if we assume that each step consists of only $Q^2/4$ loop iterations, then the iteration space of the code fragment shown in Figure 5(b) spans two steps. In this case, the data blocks accessed by the first step are 0, 4, 5, 6, and 7; and those accessed by the second step are 1, 4, 5, 6, and 7. These two sequences collectively constitute the *data-block access pattern* for this code fragment. Our ILP-based solution, presented in the next section, operates with the block-access pattern extracted from the code fragment being optimized, based on the data-block divisioning given.

It is to be noted that while the effectiveness of our energy-saving techniques is influenced by the step size and data-block size chosen, techniques for determining these sizes optimally (or near optimally) are beyond the scope of this work. Instead, we are assuming that the compiler is given a data-block size and a step size and our ILP-based techniques are applied based on these fixed sizes. However, we believe that our ILP formulations can be modified to output the optimal block/step sizes as well; we postpone this line of research to a future study. The data-access pattern extracted by the compiler is fed to the ILP solver, which is explained next.

5. ILP FORMULATION

ILP is comprised of a set of techniques that solve optimization problems in which both the objective function and the constraints are linear functions. The resulting solution variables are restricted to be integers. 0-1 ILP is a type of ILP problem in which solution variables are restricted to be either 0 or 1

Table I. Constant Terms Used in Our ILP Formulation

Constant	Definition
N	Number of memory banks
S	Number of steps
M	Number of data blocks
σ	Coefficient that captures increase in energy as a function of bank size
$R_{m,s}$	Indicates whether block m is accessed at step s
$size_{mem}$	Size of a data block
$size_{block}$	Size of a data block
$comp_{ratio}$	Compression ratio
R_{lim}	Replication limit
AE	Energy consumed by an accessed memory bank per step
IE	Energy consumed by a low-power memory bank per step
RE	Energy consumed for reactivation of a memory bank
DE	Energy consumed for deactivation of a memory bank
ME	Energy consumed for migrating a data block
$CompE$	Energy consumed for compressing a data block
$DeCompE$	Energy consumed for decompressing a data block

These constant terms are either architecture specific or program specific.

[Nemhauser and Wolsey 1988]. It is used in this article for determining the optimal bank sizes, data-to-bank mappings, data migration, data compression, and data duplication patterns. Table I gives the constant terms used in our ILP formulation. Note that these parameters are either architecture- or program specific.

Our presentation is given in five parts. In the first part (Section 5.1), we demonstrate how ILP can be used to optimally assign data blocks into memory banks. The next part (Section 5.2) partitions the given memory space into nonuniform banks in the most energy-efficient way, and determines the optimal location (bank) for each data block. The third part (Section 5.3) formulates the problem of data migration across memory banks. Finally, the fourth (Section 5.4) and fifth (Section 5.5) parts discuss the formulations of data compression and data replication, respectively, within our ILP-based framework. In all techniques presented, data-to-bank mapping is optimized as well.

5.1 Optimal Data-Block Assignment

Our objective in this section is to optimally assign[1] data blocks to (uniform) memory banks for increasing the energy benefits that could be obtained from the low-power operating modes supported by the underlying banked memory system. Based on the data-block-level access pattern extracted by the compiler (as explained in Section 4), our approach identifies the location of each data block in the memory.

We assume for now that memory space is divided into N uniform banks, namely, that all banks are of the same size. This assumption will be relaxed later in the article. Also for now, we assume we have a single low-power mode. While we give the ILP formulation based on a single low-power mode for clarity of presentation, our approach can easily be modified to work with any number

[1]We use the terms "assignment," "placement," and "mapping" interchangeably.

of low-power modes where available. Assuming that S is the number of steps (as defined earlier in Section 4) and M the number of data blocks, we can use 0-1 variables to specify the potential location (L) of each data block. Specifically, we have the following classification.

—$L_{m,n}$. This indicates whether data block m is assigned to bank n.
In our ILP formulation, we use variable A to indicate whether the bank is currently active.

—$A_{n,s}$. This indicates whether bank n is active at step s due to a read/write operation on it.
Although it is better to have a bank in low-power operating mode from the energy perspective, depending on the data-access pattern, it might be better not to go to a low-power operating mode for performance reasons. As a result, frequent switchings between low-power modes and active mode in short intervals should be avoided. Our next set of 0-1 variables are used to capture the switchings between active and low-power modes for a given bank. If a bank is activated from a low-power mode, the following variable is set to 1.

—$X_{n,s}$. This indicates whether bank n is activated at step s.
On the other hand, if a bank is switched to a low-power mode, $Y_{n,s}$ will be set as follows.

—$Y_{n,s}$. This indicates whether bank n goes to the low-power mode at step s.

After having defined our 0-1 integer variables, we can now discuss our ILP formulations. The following constraints are needed to capture the values of $X_{n,s}$ and $Y_{n,s}$. Bank n is said to be activated at step s if it is not active at step $(s-1)$ and is active at step s.

$$X_{n,s} \geq A_{n,s} - A_{n,s-1}, \quad \forall n, s \tag{1}$$

Bank n is said to be transitioned to a low-power mode at step s if it is active at step $(s-1)$ but not active at step s.

$$Y_{n,s} \geq A_{n,s-1} - A_{n,s}, \quad \forall n, s \tag{2}$$

Since a data block can reside only in a single bank at any given time, it must satisfy the following constraint.

$$\sum_{i=0}^{N} L_{m,i} = 1, \quad \forall m \tag{3}$$

The limited bank capacity forms the basis for the next constraint that needs to be included in our formulation. Assuming that the size of a block is $size_{block}$ and the available memory space is $size_{mem}$, each memory bank will be of size $size_{mem}/N$.

$$size_{block} \times \sum_{i=1}^{M} L_{i,n} \leq \frac{size_{mem}}{N}, \quad \forall n \tag{4}$$

A bank is currently active (at step s) if one of its data blocks is accessed.

$$A_{n,s} \geq R_{m,s} \times L_{m,n}, \quad \forall m, n, s \tag{5}$$

Here, $R_{m,s}$ indicates whether block m is accessed at step s. Its value is extracted from the data-access pattern. On the other hand, $L_{m,n}$ indicates whether data block m is assigned to bank n.

Having specified the necessary constraints in our ILP formulation, we next give our objective function. In our execution model, there are four components of the total memory energy consumption.

—*Active.* This is the energy consumed when a bank is in active mode.
—*Low-Power.* This is the energy consumed when a bank is in low-power mode.
—*Activation.* This is the energy consumed to activate a bank from low-power mode.
—*Deactivation.* This is the energy consumed to switch a bank to low-power mode.

Based on these components, we can express the memory energy consumption B as follows.

$$B = \sum_{i=0}^{N} \sum_{j=1}^{S} (A_{i,j} \times AE + (1 - A_{i,j}) \times IE + X_{i,j} \times RE + Y_{i,j} \times DE) \tag{6}$$

In this formulation, AE, IE, RE, and DE correspond to active energy, low-power energy, activation energy, and deactivation energy, respectively. Based on this formulation, our 0-1 ILP problem can be defined as one of minimizing B under constraints (1) through (5).

By optimally assigning data blocks to (uniform) memory banks, we are able to better exploit the available low-power operating modes. This is achieved by placing those data blocks with similar access patterns/lifetimes into the same set of banks, thereby increasing the chances for better utilizing low-power modes. Figure 6 illustrates how optimal data-block assignment could save energy in a banked memory system. In this example, we have eight data blocks (B0 through B7) and thirteen steps, and we compare the behaviors of optimal data-block assignment and the declaration-order-based assignment, namely, the case when arrays are assigned to memory banks based on their order of declaration in the program code, starting with the first location of the first bank. The bank accesses, along with the data-block accesses, are shown for convenience. Figure 6(a) depicts the data-block assignment in declaration order (assuming the data-block ids are given based on the declaration order) for a banked memory system constructed using four banks (bank 0 through bank 3). Figure 6(b) shows the potential impact of optimal data-block assignment. As can be seen from this figure, bank 0 is the first that can be placed into a low-power mode (without being reactivated). For declaration-order-based assignment, this can happen only after step 6, whereas for optimal data-block assignment, it is possible to reduce energy consumption after step 4. Similarly, after step 10, only bank 3 will be used by optimal data-block assignment. On the other hand, the declaration-order-based assignment is going to visit both bank 1 and bank 3 by

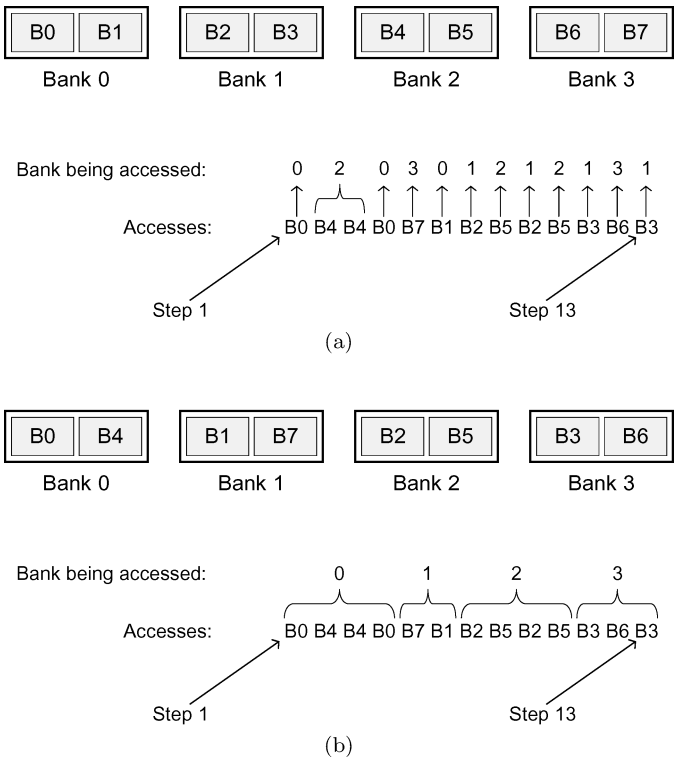

Fig. 6. Data-block layout for: (a) declaration-order-based assignment; and (b) optimal assignment.

the successive data-block accesses. This figure indicates that bank-interaccess times are highly dependent on data-block assignment, which could affect the energy consumption dramatically.

Discussion. Recall that our ILP formulation so far is based on a single low-power mode. Instead of using a single low-power mode as done previously, we could also use multiple low-power modes. In this case, given a bank-interaccess time, our approach selects the most suitable power mode to use, and each memory bank can be in one of the four modes listed in Figure 2 (i.e., active, standby, nap, power-down) at any given time. To accommodate multiple low-power operating modes in our ILP formulation, we define $B_{S_{n,s}}$, $B_{N_{n,s}}$, and $B_{P_{n,s}}$ for standby, nap, and power-down modes, respectively. These variables are set to 1 if bank n is in the corresponding mode at step s. To ensure a unique mode selection for each bank at every step (s), the following constraint is included.

$$A_{n,s} + B_{S_{n,s}} + B_{N_{n,s}} + B_{P_{n,s}} = 1, \quad \forall n, s \tag{7}$$

Note that we assume the operating-mode transitions can occur only between the active mode and a low-power operating mode. For each low-power operating mode, the corresponding mode transitions have to be identified. Recall that $X_{i,j}$ and $Y_{i,j}$ are defined for this purpose. We define $X_{S_{i,j}}$, $X_{N_{i,j}}$, and $X_{P_{i,j}}$ for transitions from, respectively, the standby, nap, and power-down modes to the

active mode. Similarly, $Y_{S_{i,j}}$, $Y_{N_{i,j}}$, and $Y_{P_{i,j}}$ are used for capturing deactivations. Based on these definitions, we have the following new constraints. For the standby mode, we have

$$X_{S_{n,s}} \geq A_{n,s} + B_{S_{n,s-1}} - 1, \quad \forall n, s. \tag{8}$$

$$Y_{S_{n,s}} \geq A_{n,s-1} + B_{S_{n,s}} - 1, \quad \forall n, s. \tag{9}$$

For the nap mode, we have

$$X_{N_{n,s}} \geq A_{n,s} + B_{N_{n,s-1}} - 1, \quad \forall n, s. \tag{10}$$

$$Y_{N_{n,s}} \geq A_{n,s-1} + B_{N_{n,s}} - 1, \quad \forall n, s. \tag{11}$$

For the power-down mode, we have

$$X_{P_{n,s}} \geq A_{n,s} + B_{P_{n,s-1}} - 1, \quad \forall n, s. \tag{12}$$

$$Y_{P_{n,s}} \geq A_{n,s-1} + B_{P_{n,s}} - 1, \quad \forall n, s. \tag{13}$$

Finally, the memory energy consumption B in expression (6) should be rewritten as follows.

$$\begin{aligned} B = & \sum_{i=1}^{N} \sum_{j=1}^{S} CA_{i,j} \times AE + (A_{i,j} - CA_{i,j}) \times NE \\ & + B_{S_{i,j}} \times IE_S + B_{N_{i,j}} \times IE_N + B_{P_{i,j}} \times IE_P \\ & + X_{S_{i,j}} \times RE_S + X_{N_{i,j}} \times RE_N + X_{P_{i,j}} \times RE_P \\ & + Y_{S_{i,j}} \times DE_S + Y_{N_{i,j}} \times DE_N + Y_{P_{i,j}} \times DE_P \end{aligned} \tag{14}$$

Note that in the aforesaid formulation, low-power energy IE, activation energy RE, and deactivation energy DE are replaced with their counterparts for each operating mode. For example, low-power energy IE is replaced with IE_S, IE_N, and IE_P. After these modifications/enhancements, our 0-1 ILP problem under multiple low-power modes can be defined as one of minimizing B, under constraints (3) through (5) and (7) through (13).

5.2 Partitioning Memory Space into Nonuniform Banks

The energy consumption of the memory system can be further reduced by partitioning the available memory space into nonuniform (sized) banks. We determine the number of banks and their sizes based on the data-access pattern and total memory size (to be partitioned) using 0-1 variables. For each possible bank size, we define 0-1 variables. By using these 0-1 variables, we then determine the partitions (banks) and their contents. Although we restrict the possible sizes to be powers of two to simplify the problem, this can be extended to cover other possible sizes as well. This way, we keep the problem size smaller (fewer variables and constraints) and by doing so reduce the ILP solution time. If, for example, the memory size in question is $4k$ (assuming that k is a power of two) and the minimum bank-size possible is k, then we can potentially have 0-1 variables for one $4k$-bank, two $2k$-banks, and four $1k$-banks. If two of the four $1k$-bank variables and one $2k$-bank variable are returned as 1 from the ILP solver, then we conclude that the application under consideration will spend the minimum memory energy when the memory space is partitioned into three

banks of sizes k, k, and $2k$. Our formulation also gives the optimum mapping of the data blocks to these banks, as in the previous section.

To incorporate nonuniform banks into our ILP formulation, several modifications to the original problem have to be made. In this part of our discussion, we denote a memory bank using a pair of attributes (l, n), where l is used for its size (in terms of data blocks it can hold) and n is used for distinguishing one bank from others that have the same size. For example, with a memory of size $8k$, there can be two banks with a size of $4k$, where k is the minimum bank-size possible. These two banks can be expressed using the pairs $(4k, 1)$ and $(4k, 2)$.

Assuming that N denotes the maximum number of banks possible if each bank holds only one data block (which occurs when the minimum possible bank size is used), S is the number of steps, and M the number of data blocks, we redefine the 0-1 variables used to specify the potential location L of each data block. Specifically, we have the definitions next given.

—$L_{m,l,n}$. This indicates whether data block m is assigned to bank (l, n). There are possibly $\frac{N}{l}$ banks of size l ($\frac{N}{l}$ at most, if all banks are of size l). We use a variable for each one of these bank candidates. If this 0-1 variable is 1, then we conclude that the corresponding bank actually exists (i.e., our ILP-based solver returns a memory-space partitioning that contains such a bank). In addition to the modification in location variable, we also introduce a new 0-1 variable. We specify whether bank (l, n) actually exists (returned by our ILP formulation) by using $E_{l,n}$. In other words, we have the next definition.

—$E_{l,n}$. This indicates whether data bank (l, n) exists.

Since the way we identify the memory banks is different from that employed in Section 5.2, we need to replace some of the variables and constraints as well. Specifically, we replace the activation and deactivation variables $X_{n,s}$ and $Y_{n,s}$ with $X_{l,n,s}$ and $Y_{l,n,s}$, respectively, as well as the corresponding constraints. Consequently, expressions (1) and (2) should be replaced with expressions (16) and (16).

$$X_{l,n,s} \geq A_{l,n,s} - A_{l,n,s-1}, \quad \forall l, n, s \tag{15}$$

$$Y_{l,n,s} \geq A_{l,n,s-1} - A_{l,n,s}, \quad \forall l, n, s \tag{16}$$

As mentioned earlier, we assume bank sizes are restricted to be powers of two (though our formulation can be modified to drop this requirement). As an example for the sake of illustration, if the memory in question can hold at most 8 data blocks, then this memory can be partitioned into memory banks such that each bank can hold 1, 2, 4, or 8 data blocks. Since the available memory space is partitioned among banks, the total size of the banks (determined by the ILP solver) should be equal to the available memory space.

$$\sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} E_{2^i,j} \times 2^i = size_{mem} \tag{17}$$

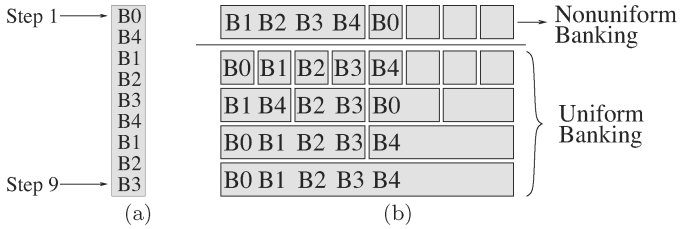

Fig. 7. Data-block layout for nonuniform banking and uniform banking with fixed bank-sizes of 1, 2, 4, and 8.

A data block can be residing in a bank only if the bank in question exists.

$$E_{l,n} \geq L_{m,l,n}, \quad \forall m, l, n \tag{18}$$

The unique location constraint given earlier by expression (3) has to be adjusted as well.

$$\sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} L_{m,2^i,j} = 1, \quad \forall m \tag{19}$$

Since banks are not uniform, the sizes may differ. Specifically, instead of expression (4), the following expression can be written for each possible bank size l.

$$size_{block} \times \sum_{i=1}^{M} L_{i,l,n} \leq l, \quad \forall l, n \tag{20}$$

Similar to expression (5), a bank is currently active (at step s) if one of its data blocks is accessed.

$$A_{l,n,s} \geq R_{m,s} \times L_{m,l,n}, \quad \forall m, l, n, s \tag{21}$$

As before, $R_{m,s}$ (extracted from the data-access pattern given by the compiler) indicates whether data block m is accessed at step s. Based on the preceding modifications, our objective function B is redefined (to include all sources of energy consumption) as follows.

$$B = \sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} \sum_{k=1}^{S} (A_{i,j,k} \times AE + (1 - A_{i,j,k}) \times IE + X_{i,j,k} \times RE + Y_{i,j,k} \times DE) \times \sigma^i \tag{22}$$

Note that since we do not have a unique bank size, σ is used to capture the energy consumption behavior of banks with different sizes (capacities). For example, if the size of bank is doubled, the energy spent would be σ times the original energy consumption. Based on this formulation, our 0-1 ILP problem can be defined as one of minimizing B under constraints (15) through (21).

To better explain how our approach partitions the available memory space into nonuniform banks and why one can expect benefits from it, we give the banks and data blocks of an example application in Figure 7. In this example,

we have five data blocks (B0 through B4) and nine steps, and we contrast the behaviors of nonuniform banking and the uniform banking with fixed bank-sizes of 1, 2, 4, and 8. In Figure 7(a) the data-access pattern is shown and Figure 7(b) shows how each approach places the data blocks into the banks. One can see from this figure that our scheme clusters those data blocks accessed together frequently; that is, B1, B2, B3, and B4 are placed into the same bank (a bank that can hold four data blocks). As for the uniform scheme, the smaller bank sizes (e.g., 1) enable to use only a small portion of the memory to save energy; however, frequent activations/deactivations resulting from the access pattern might cause even more energy consumption. Similarly, having a memory bank with only a small portion used wastes energy compared to a perfect fit. In comparison, by making use of nonuniform memory banks, our approach is able to cluster those data blocks that are accessed together and place them into the most suitable-sized banks.

Discussion. We now discuss two modifications to the problem defined in the previous section. First, we relax the assumption that suggests memory banks be only powers of two. Second, we introduce an additional constraint to put a limit on the performance overhead due to memory banking.

Recall that memory-bank sizes are assumed to be powers of two. If we were to relax this assumption so that we could use all possible bank sizes (not only powers of two), Eq. (17) has to be replaced with the following.

$$\sum_{i=1}^{N} \sum_{j=1}^{\frac{N}{i}} E_{i,j} \times i = size_{mem} \tag{23}$$

Similarly, the start and end values of index variables i and j in Eqs. (19) and (22) have to be changed in the same manner as well. For example, if we assume that the memory can hold eight data blocks, banks with powers of two can have sizes of (1,2,4,8), whereas the banks with any size can have sizes of (1,2,3,4,5,6,7,8). Note that the total memory size is kept constant and the only difference is in the way the available memory space is partitioned across banks.

So far in our discussion we have not put any limit on the potential performance degradation due to turning on/off the memory banks. One might envision a case where only a limited degradation in performance could be tolerated. The performance overhead in our formulation can be captured using an additional constraint. In our approach, the performance degradation incurred is mainly due to two factors: bank reactivation and bank deactivation, captured by $X_{l,n,s}$ and $Y_{l,n,s}$, respectively. Assuming that O_{max} is the maximum performance overhead allowed for the design (which can be 0 to obtain the best energy savings without tolerating any performance penalty), then our performance constraint can be expressed as follows.

$$O = \sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} \sum_{k=1}^{S} (X_{i,j,k} \times RP + Y_{i,j,k} \times DP) \leq O_{max}. \tag{24}$$

In the aforesaid expression, RP and DP are used to capture the performance overheads incurred due to activating and deactivating, respectively, a memory bank.

It should be noted that, for multiple low-power modes, the same modifications described in the uniform-banking case in the previous section have to be employed. In other words, in addition to $A_{l,n,s}$, three low-power operating modes ($B_{S_{l,n,s}}$, $B_{N_{l,n,s}}$, and $B_{P_{l,n,s}}$) need to be defined. Moreover, as in the case of uniform banking, activation/deactivation variables for different low-power modes should be included. Specifically, we define $X_{S_{i,j,k}}$, $X_{N_{i,j,k}}$, and $X_{P_{i,j,k}}$ for transitions from, respectively, the standby, nap, and power-down modes to the active mode. Similarly, $Y_{S_{i,j,k}}$, $Y_{N_{i,j,k}}$, and $Y_{P_{i,j,k}}$ are used for deactivation.

5.3 Data Migration

Further energy improvements can be achieved by migrating data blocks among banks during the course of execution. In particular, in some cases, instead of activating/deactivating a bank, it might be more beneficial to transfer the data block to an already-active bank. To incorporate data migration into our ILP formulation, several modifications to the original problem have to be made. An important point we would like to make clear is that the migration decisions in our approach are taken at compile time (by the ILP solver); however, the migrations themselves take place at runtime.

Since the location of a data block is no longer restricted to be the same throughout the execution of the program (due to migration), we have to include the step parameter s in our location variable L. For this purpose, $L_{m,l,n}$ is replaced with $L_{m,l,n,s}$ to indicate whether block m is residing in bank (l, n) at step s.

Consequently, expressions (18), (19), and (20) given previously should be replaced with expressions (25), (26), and (27), respectively, to capture the bank-existence constraint, data-block-location constraint (a data block can be in a single bank at any time), and bank-size constraint.

$$E_{l,n} \geq L_{m,l,n,s}, \quad \forall m, l, n, s \tag{25}$$

$$\sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} L_{m,2^i,j,s} = 1, \quad \forall m, s \tag{26}$$

$$size_{block} \times \sum_{i=1}^{M} L_{i,l,n,s} \leq l, \quad \forall l, n, s \tag{27}$$

Eq. (21), on the other hand, does not require any change except for the variable renaming, since it is already subscripted by step parameter s.

$$A_{l,n,s} \geq R_{m,s} \times L_{m,l,n,s}, \quad \forall m, l, n, s \tag{28}$$

Migration behavior of data blocks needs to be captured as well. In our formulation, we use $Z_{m,s}$ for this purpose. A data block m migrates (at step s) if its

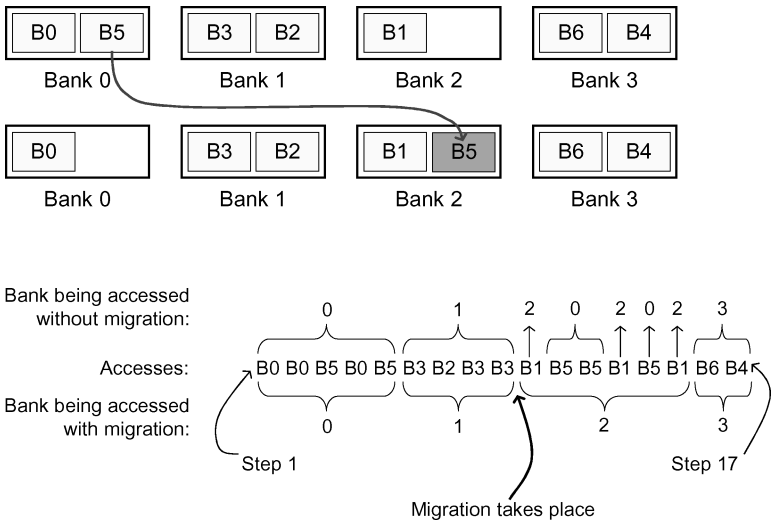

Fig. 8. A sample scenario that illustrates the potential benefits of data migration. Initial state and state after migration are shown.

current location (bank) is different from its previous location (i.e., the one in the previous step).

$$Z_{m,s} \geq L_{m,l,n,s} - L_{m,l,n,s-1}, \quad \forall m, l, n, s \tag{29}$$

Migration typically brings additional performance overheads on top of the bank-related overheads. To capture this additional overhead, the constraint in expression (24) should be replaced with the following.

$$O + \sum_{i=1}^{M} \sum_{j=1}^{S} (Z_{i,j} \times MP) \leq O_{max} \tag{30}$$

In the preceding expression, MP is used to capture the migration cost for a data block, whereas O reflects the original overhead given in expression (24). Note that, conservatively, we assume migrations cannot be overlapped.

Although data migration can reduce the energy consumption (as it can empty banks which can be placed into low-power mode), it also causes extra energy consumption since the migration itself consumes energy. This overhead has to be included in the ILP formulation.

$$T = \sum_{i=1}^{M} \sum_{j=1}^{S} (Z_{i,j} \times ME) \tag{31}$$

Then, our new 0-1 ILP problem can be defined as minimizing $B + T$ under constraints (15) through (17) and (25) through (30).

Data migration moves data from one memory bank to another at runtime, in an attempt to better exploit available low-power operating modes. This is possible since migration can place data blocks with similar access patterns/lifetimes into the same set of banks, thereby increasing the chances for better utilizing low-power modes. Figure 8 illustrates, using an example, how data migration

could save energy in a banked memory system. The layout given on the top in Figure 8 depicts the initial state for a banked memory system constructed using four banks (bank 0 through bank 3). It also shows the layout before the migration. The data-access pattern is given on the bottom of the figure, along with bank accesses for the migration and no-migration cases. By using migration, it is possible to place bank 0 into low-power operating mode after step 5, whereas this is delayed until the end of step 14 if migration is not used. As can be seen from Figure 8, data block 5 is migrated from bank 0 to bank 2, allowing bank 2 to service the requests for data block 5.

5.4 Data Compression

We now discuss our next technique for increasing the energy benefits that could be obtained from low-power operating modes: data compression. Using data compression, a memory bank can hold more data blocks. This in turn can reduce the number of banks active during execution, thereby reducing the memory energy consumption further. In order to incorporate data compression/decompression into our ILP formulation, extra variables, constraints, and energy costs have to be defined. In addition to our state variable $L_{m,l,n,s}$, we define $C_{m,l,n,s}$ to distinguish between compressed and uncompressed data blocks.

—$C_{m,l,n,s}$. This indicates whether data block m is compressed and in bank (l, n) at step s.

Similar to an uncompressed data block (given by expression (25) earlier), a compressed data block can reside in a bank only if the bank in question exists.

$$E_{l,n} \geq C_{m,l,n,s}, \quad \forall m, l, n, s \tag{32}$$

Since a data block can reside only in a single bank at any given time, it must be in one of the memory banks in the compressed or uncompressed form. Specifically, we replace expression (26) with the following constraint.

$$\sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} L_{m,2^i,j,s} + C_{m,2^i,j,s} = 1, \quad \forall m, s \tag{33}$$

Expression (27) has to be adjusted as well, since compressed data blocks consume less space. The size of a compressed data block is $comp_{ratio} \times size_{block}$. Consequently, the following expression can be written for each possible bank size l.

$$size_{block} \times \left(\sum_{i=1}^{M} L_{i,l,n,s} + comp_{ratio} \times \sum_{i=1}^{M} C_{i,l,n,s} \right) \leq l, \quad \forall l, n, s \tag{34}$$

Here, we are assuming that a data block can be accessed only if not in compressed form (i.e., in order for an access to take place, the data block in question needs first to be decompressed). Depending on the data-access pattern, the $C_{m,l,n,s}$ values for these data blocks at the corresponding steps will be set to 0.

$V_{m,s}$ and $W_{m,s}$ are used to express compression and decompression (of block m at step s), respectively.

$$V_{m,s} \geq C_{m,l,n,s} + L_{m,l,n,s-1} - 1, \quad \forall m, l, n, s \tag{35}$$

$$W_{m,s} \geq L_{m,l,n,s} + C_{m,l,n,s-1} - 1, \quad \forall m, l, n, s \tag{36}$$

Compressions and decompressions typically bring additional overheads on top of the migration-related overheads. We can express the additional energy consumption due to compression/decompression.

$$C = \sum_{i=1}^{M} \sum_{j=1}^{S} (V_{i,j} \times CompE + W_{i,j} \times DeCompE) \tag{37}$$

In the previous expression, *CompE* and *DeCompE* are used to capture compression and decompression costs for a data block, respectively. Finally, based on expression (37), our ILP formulation can be reexpressed as one of minimizing $B + T + C$.

Data compression squeezes data blocks in memory, which in turn allows better use of the available DRAM space, and makes it possible to increase the number of inactive (idle) banks which are candidates for being put in low-power operating modes. Figure 9 illustrates how data compression can save energy in a banked memory system. Figure 9(a) shows the initial state of a banked memory system with four banks (bank 0 through bank 3). If an application accesses those blocks captured by the data-access pattern from these banks, bank 0 and bank 1 will be interleavingly accessed between steps 6 to 14 by the application. Figure 9(b) shows the impact of using data compression. In this example, data block 3 is compressed and moved (i.e., migrated) from bank 1 to bank 0. Also, data block 1 is compressed to create sufficient space for block 3. This allows us to put bank 1 in low-power mode until step 14 (note that in this particular example, if there were sufficient empty space in bank 0, we could have migrated all the blocks to it without using any compression).

Another point is that if a decompression algorithm takes too much time, the main impact on our approach would be that of scheduling predecompressions earlier. Of course, depending on the latency of the particular decompression algorithm at-hand, if the predecompressions have to be scheduled very early (to minimize their potential impact on performance), the memory-saving benefits of our approach will reduce. In the extreme case, we may decide that using a particular compression algorithm does not make sense due to its very high latency; that is, we may not be able to schedule predecompressions. Nonetheless, our preliminary analysis of known algorithms, such as LZW and zero-removal, shows our approach is able to schedule decompressions ahead of time to hide their latencies, while at the same time achieving large memory-savings.

5.5 Data Replication

The last energy-saving technique we discuss in this article is data replication. The main objective of this technique is to replicate data blocks to minimize the overall energy consumption; in other words, to maximize energy the savings that can be obtained from low-power operating modes. In this section,

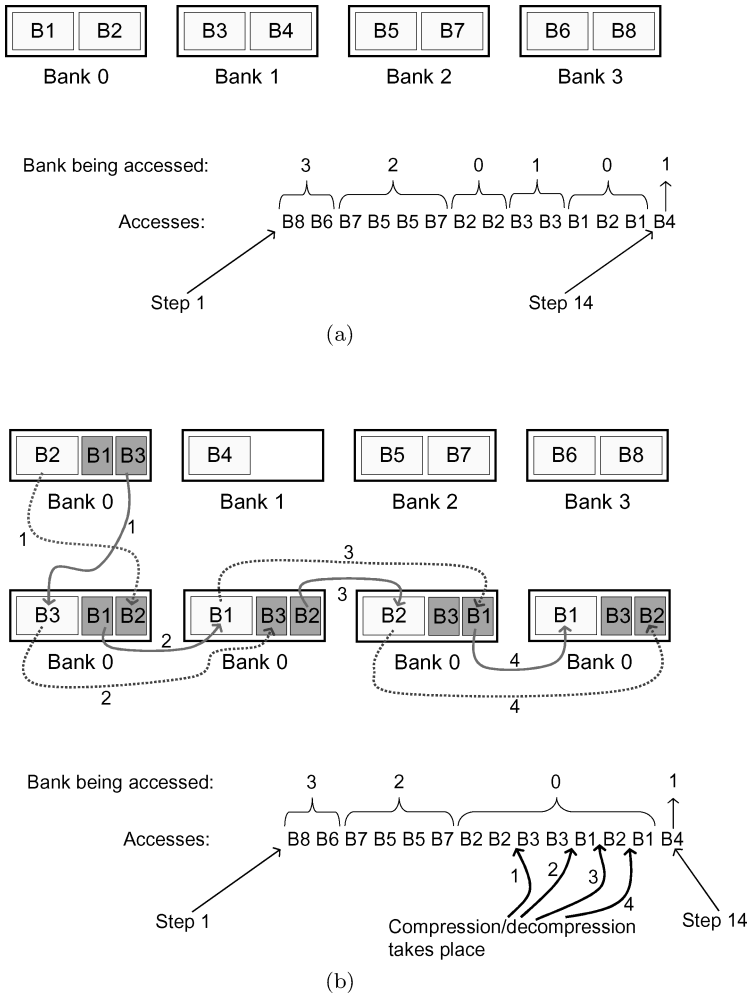

Fig. 9. A sample scenario that illustrates the potential benefits of data compression: (a) initial state; (b) state after compression.

we do not restrict the location of a data block to a single bank in order to allow data replication. A data block m must be in at least one of the banks at each step s. This is captured by replacing expression (33) with the following constraint.

$$\sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} L_{m,2^i,j,s} + C_{m,2^i,j,s} \geq 1, \quad \forall m, s \tag{38}$$

Note that the only change to this constraint is the use of the $\geq$ operator, instead of the $=$ operator. The replication limit (R_{lim}), set by the designer, bounds the

number of blocks with replicas. In mathematical terms, we have

$$\sum_{i=1}^{M} \sum_{j=0}^{\log_2 N} \sum_{k=1}^{\frac{N}{2^j}} L_{i,2^j,k,s} \leq M + R_{lim}, \quad \forall s. \tag{39}$$

In the previous expression, the total number of data blocks in the memory banks is obtained by the sum expression on the left side. This sum should be less than or equal to the number of data blocks plus the number of replicas ($M + R_{lim}$).

If a data block is being accessed at a certain step, then this access request should be serviced by one of the memory bank(s) in which this block is residing. Recall that in order to ensure this, we have defined the 0-1 variable $A_{l,n,s}$. This variable takes a value of 1 if bank (l, n) is active, that is, a bank is considered as currently active (at step s) if one of its data blocks is accessed. However, this is now no longer valid, since there might be multiple copies of a data block. Consequently, we need to employ an additional variable $BB_{m,l,n,s}$ to indicate whether data block m is being accessed at step s and resides in bank (l, n), which is active.

$$BB_{m,l,n,s} \leq L_{m,l,n,s}, \quad \forall m, l, n, s \tag{40}$$

$$BB_{m,l,n,s} \leq A_{l,n,s}, \quad \forall m, l, n, s \tag{41}$$

Expression (41) captures whether the data block is located in the corresponding bank and expression (41) ensures the bank is active.

$$\sum_{i=0}^{\log_2 N} \sum_{j=1}^{\frac{N}{2^i}} BB_{R_{m,s},i,j,s} \geq 1, \quad \forall m, s \tag{42}$$

In the preceding equation, $R_{m,s}$ indicates whether block m is accessed at step s (its value is extracted from the data-access pattern). This indicates an access to the data block and hence, one of the banks in which this block resides should be active during this step. The sum of banks that satisfy this constraint should be greater than or equal to 1.

To illustrate our point, let us consider the scenario depicted in Figure 10. In this scenario, we assume that an application program makes 17 accesses (to data blocks in the order given on the bottom of the figure) to the memory system which is partitioned into four banks ($B_0 \ldots B_3$). The first layout on the top shows the case where no replication is used, and the corresponding bank accesses are depicted above the data-access pattern. Now, consider the alternate storage scenario shown on the bottom of the figure, where a replica of data block 5 is stored in bank B_2; that is, data block 5 is replicated across two banks. The bank-access pattern in this case is illustrated below the data-block accesses. Comparing this with the previous bank-access pattern, it can be seen that the one employing replication is preferable from the low-power-management viewpoint, since it increases the period during which a bank is idle, namely, it increases bank-interaccess time. By using replication, it is possible to place bank 0 into low-power operating mode after step 5, whereas this is not possible

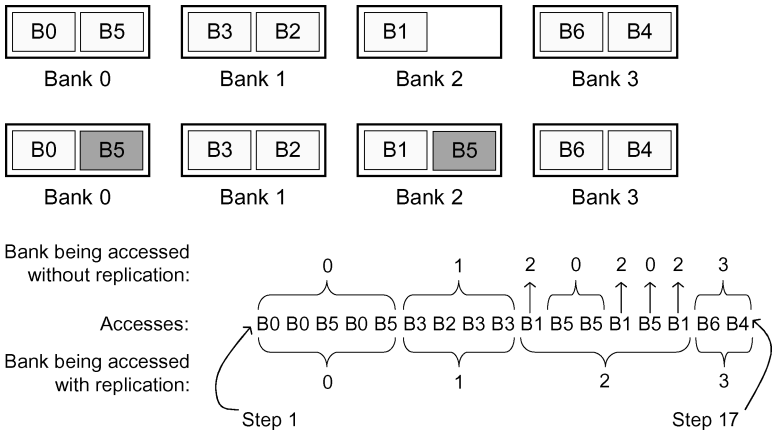

Fig. 10. An example scenario with seven data blocks and four banks, illustrating data replication. The block accessed at each step (data-access pattern) is given on the bottom.

before step 14 if replication is not adopted. This small example shows how data replication can help reduce energy consumption in a banked memory system.

5.6 Instruction Accesses

Although our main goal is to reduce the energy consumption within data accesses, this approach can easily be modified to target instruction accesses. For this purpose, an optimizing compiler can analyze the loop-intensive applications and employ an instruction block (instruction tile) approach similar to that of a data block. Specifically, the unit of instructions being stored in banks is denoted by an instruction block. Due to the sequential nature of instruction accesses, the benefits brought by this approach for instruction accesses may not be as substantial as for the case of data accesses. This follows from the fact that instruction accesses usuallys access the banks in sequential fashion, which utilizes bank accesses and increases the bank idleness period. Apart from this difference, most of the aforementioned formulation targeting data accesses can be modified to work for an instruction-access pattern.

6. CODE MODIFICATION

As shown in Figure 1, the last component of our approach is a code modification module. The purpose of this component is to modify the application source-code by inserting explicit calls to manage data-block activity within the banked memory system. Since modifications for the different techniques discussed in this work are similar, we illustrate code modification only for data migration.

Our compiler inserts migration management code at each loop nest that uses migration. Figure 11 shows an example loop nest from one of our benchmarks, as well as the transformed loop nest augmented with the memory-bank management code and data-block assignments. In this example, there are two arrays, X and Y, each with N^2 elements. We assume that each array is divided into four

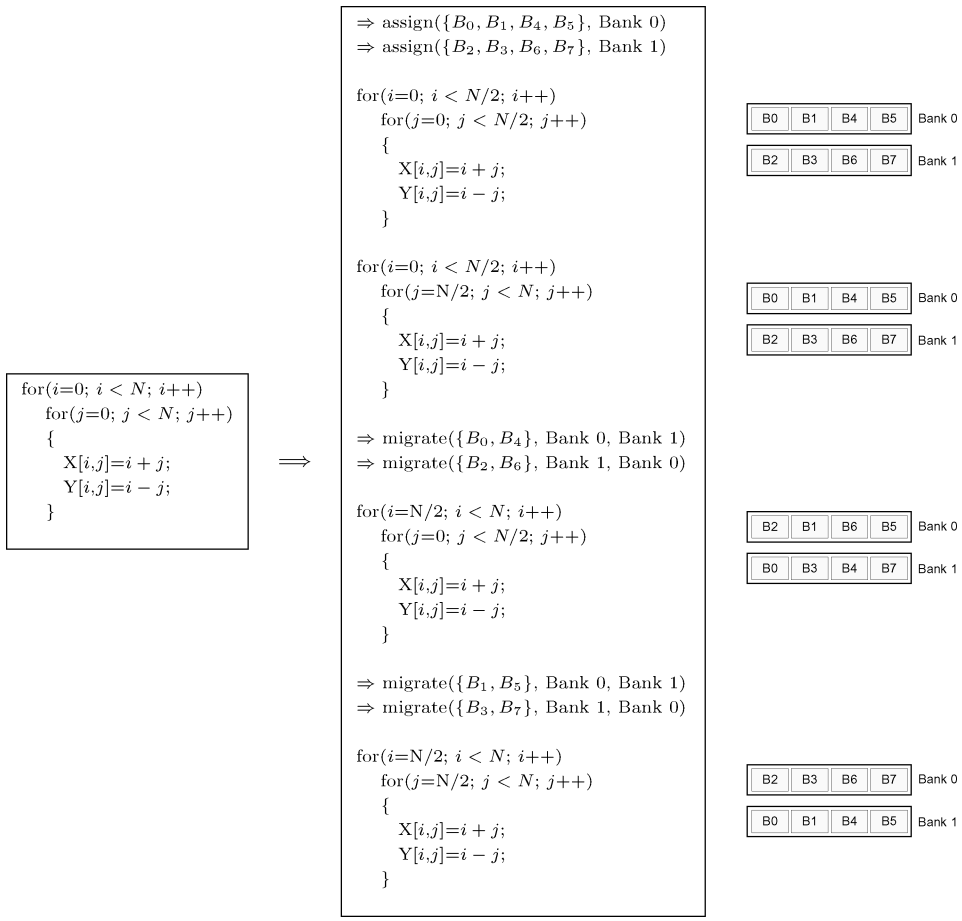

Fig. 11. An example loop nest from one of our benchmarks, written in C (left) and the transformed loop nest augmented with the memory-bank-management code (middle). Lines marked with "⇒" are inserted by our compiler. The loop nest is accessed in four steps, each having $N^2/4$ iterations, and the transformed code is structured based on the number of steps. The right part of the figure illustrates bank contents at different points.

equal data blocks. Specifically, $B_0 = X[0..(\frac{N}{2}-1), 0..(\frac{N}{2}-1)]$, $B_1 = X[0..(\frac{N}{2}-1), \frac{N}{2}..(N-1)]$, $B_2 = X[\frac{N}{2}..(N-1), 0..(\frac{N}{2}-1)]$, $B_3 = X[\frac{N}{2}..(N-1), \frac{N}{2}..(N-1)]$. Similarly, B_4, B_5, B_6, and B_7 correspond to the data blocks of Y. We are assuming the original loop nest is accessed in four steps, each having $N^2/4$ iterations. In the transformed code, we use "⇒" to indicate the lines inserted by our compiler. Before entering the loop nests, we need to make the initial assignments of data blocks (to banks). This is expressed in the transformed code by the *assign(block, bank)* command. The first parameter of this command consists of the block id(s) and the second parameter is the bank in which this data block will (initially) be stored. Migration decisions are expressed by the *migrate(block, source, destination)* command.

Table II. Benchmark Codes Used in This Study

Benchmark Name	Source	Total Data Size (KB)	Data Block Size (KB)	Number of Arrays	Executable Size (KB)
adi	Livermore	468.8	39.1	6	9.5
apsi	Spec	78.2	19.5	17	9.8
bmcm	Perfect Club	234.4	19.5	11	9.5
btrix	Spec	75.0	11.7	29	18.2
mxm	Spec	937.6	78.0	3	7.0
tomcatv	Spec	546.9	19.5	9	13.7
wss	Perfect Club	70.8	17.7	10	9.3

7. EXPERIMENTAL EVALUATION

7.1 Setup

The necessary code analysis (to extract access patterns) and modifications (to insert explicit migration, compression/decompression, and replication calls in the code) are automated within the SUIF compiler [Wilson et al. 1994], and the ILP solver used for implementation is XPress-MP [XPress 2002], a commercial tool. To test the effectiveness of our ILP-based approach in reducing memory energy consumption, we performed several experiments with seven array-intensive benchmark codes, using a simulation environment built upon SIMICS [Magnusson et al. 2002]. Table II lists the benchmark codes (taken from Livermore, Spec, and Perfect Club benchmark suites) used in this study and their important characteristics. While these codes originally used floating-point data, we converted them to integer data to test our compression scheme. The third column of this table gives the total size of the data processed by each benchmark, and the fourth column shows the data-block size used. The fifth column gives the number of arrays accessed by each benchmark. The last column shows the size of each benchmark. The default simulation parameters used in our experiments are given in Table III. The low-power energy values given in Table III reflect the actual energy consumption values from Figure 2. We additionally assume that read/write incurs extra energy consumption in the active mode. During activation (from low-power mode to active mode) and deactivation (from active mode to low-power mode), a full active-mode energy is assumed as spent (to be conservative).

Note that energy consumption for an access is due to accessing the whole memory bank. However, the compression energy is only due to compressing a single data block. Moreover, read-access energy is not captured within the compression energy cost, but rather has been captured separately as an individual access. In expression (15), $CA_{i,j}$ indicates whether bank i is active at step j, meaning that it is being accessed ($A_{i,j}$ = active, $CA_{i,j}$ = active and accessed). A bank will be active if a compression occurs, that is, $CA_{i,j} = 1$. Therefore, an active-bank energy will be incurred while compression energy is also included.

In Table III we report our results based on a fixed compression ratio given by $\sim$2. Our goal by assuming a constant compression ratio is to show that compression can be a useful technique to consider in banked memory architectures. Alternatively, in one of our previous works [Ozturk et al. 2006], we discussed

Table III. Default Simulation Parameters

Parameter	Value
Number of Banks	4
σ	1.3
Compression Ratio	∼2
Replication Limit	20%
Technology	0.35 micron
Processor	
Processor type	500 MHz, 2-issue
I-Cache	32KB, Direct mapped
D-Cache	32KB, Direct mapped
Per Access Read Energy for Cache	0.20 nJ
Per Access Write Energy for Cache	0.21 nJ
Bus	
On-Chip Bus Transaction Energy	0.04 nJ
Off-Chip Bus Transaction Energy	3.48 nJ
Per Cycle Clock Energy	0.18 nJ
Memory	
Data Block Size: Memory Bank Capacity	1:2
Power Down Bank Energy	0.005 nJ
Active Bank Energy	3.570 nJ
Non-accessed Active Bank Energy	3.500 nJ
Standby Bank Energy	0.830 nJ
Nap Bank Energy	0.320 nJ
Bank Activation Energy	3.570 nJ
Bank Deactivation Energy	3.570 nJ
Data Migration Energy	3.570 nJ
Data Compression Energy	2.668 nJ
Data Decompression Energy	2.668 nJ

how a compression-based approach can be implemented when the different blocks can have different compression ratios. If desired, that approach can also be used in our implementation to operate with different block sizes.

In this approach, the program starts its execution with all of its tiles compressed. A compressed tile is decompressed and stored in the decompression buffer by a *decompressor* before it is accessed by the program. The important point to emphasize here is that this approach is not tied to any specific compression/decompression algorithm, and the compressor and decompressor can be implemented either in software or hardware. Compressed tiles are stored in the compressed area. The memory in this area is divided into equal-sized *slices*. The size of a slice is smaller than that of a block in the decompression buffer. Although the size of tiles is constant, the compression ratio depends on the specific tile. Therefore, the number of slices required to store a compressed tile may vary from one tile to another, hence, slices of the same tile form a link table. This way, it is possible to accommodate different compression ratios.

For each benchmark code listed in Table II, we performed experiments with five different versions and their different combinations.

—*Optimal Data Placement (OD)*. This is the classical uniform banked memory management strategy that does not use any data migration, data compression, or data replication. Data blocks are placed in memory banks and do

not move during the course of execution. However, it should be emphasized that, apart from migration, compression/decompression, and replication, this version makes *full use of the low-power modes available and data placement is optimal*.

—*Nonuniform Banking (NU)*. This is the integer-linear-programming-based strategy discussed in this article, wherein banks with different sizes are employed. Those data blocks frequently accessed together can be put in a larger bank, whereas a single data block (without any access-pattern relationship with other data blocks) can be placed into a smaller-sized bank to optimize the energy consumption in active banks and during bank activations/deactivations. Note that data blocks are optimally placed in memory banks, that is, this is an extension of the previous scheme (i.e., OD).

—*Migration (ME)*. This is also an extension to OD (see Section 5.2), where data migration is employed to further decrease the energy consumption. An accessed data block is migrated to an active bank if so doing is profitable from the energy consumption point-of-view. The rationale behind this scheme is to transfer the data block being accessed to one of the active memory banks, thereby reducing the number of active memory banks as much as possible.

—*Compression (CO)*. This scheme also extends OD (see Section 5.3), wherein data-block compression is employed to decrease the number of banks occupied by data. Compression will provide additional space that can be used for accommodating additional data blocks in the same bank.

—*Replication (RE)*. In this version, data blocks are placed into banks and replicated based on the access pattern exhibited by the application. Data replication decisions are based on the values extracted from the integer linear programming formulations, and give the optimum result. Again, data blocks are optimally placed in memory banks, as in OD.

In the experimental results, to be presented shortly, the term "energy consumption" is used to refer to that energy expended in the memory banks (during data migrations, data compressions, and data accesses, as well as that consumed during compression, decompression, and migration). Since our focus is on memory energy consumption due to data accesses, the experimental results presented in the following are for this energy consumption. However, enhancing SIMICS using Wattch-like [Brooks et al. 2000] energy models, we also measured the energy impact of our approach on other system components. For this purpose, we assumed that program instructions are stored in two reserved memory banks and that there is an on-chip cache of 32KB. We found the memory energy due to data accesses to be the dominating element in these benchmarks and to account for nearly 34.2% of the overall energy consumption. Therefore, one can expect significant savings in reducing memory energy consumption due to data accesses. However, our optimizations also incur some additional energy consumption due to code modifications, compressions, decompressions, and migration. We found that, except for the energy due to code modifications (primarily loop tiling), the total contribution of the other types of overheads is about 1% and these are included in all the results presented next. The overheads due

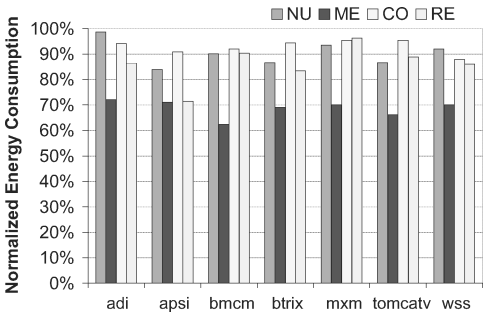

Fig. 12. Normalized energy consumption values with respect to the OD version.

to tiling, on the other hand, are not included in the results, as tiling is applied as a preprocessing step. Tiling-related overheads are found to be around 2.6% when averaged over all benchmarks in our experimental suite.

7.2 Baseline Results

Figure 12 shows the normalized energy consumption values with NU, ME, CO, and RE over the OD version. We see that the average reduction in energy consumption is 9.8%, 31.3%, 7.2%, and 13.9%, respectively, for NU, ME, CO, and RE. One can make several observations from these results. First, OD generates the worst results for all seven benchmarks in our experimental suite, mainly because of its large number of active banks at any point during execution. Our second observation is that ME generates the best results for all the benchmarks tested. This is due to the fact that data migration moves data from one memory bank to another at runtime, which enables to better exploit the available low-power operating modes. By placing data blocks with similar access patterns/lifetimes into the same set of banks, we increase the chances for better utilizing low-power modes. The solution times incurred with the ILP-based approach are not excessive. Specifically, the largest solution times when using the NU, ME, CO, and RE schemes are 18.2, 55.4, 137.5, and 244.1 minutes, respectively. Our belief is that these solution times are within tolerable range, particularly for embedded systems where one can invest a large number of cycles in compilation, as the code quality is of utmost importance. It need also be mentioned that when we consider the overall overheads incurred by the entire framework shown in Figure 1, the ILP solver is certainly the most time-consuming component. In comparison, the additional overheads incurred by the compiler when using all optimizations simultaneously are very small.

Note also that, as mentioned earlier, since data-memory energy originally constitutes about 34.2% of the overall energy consumption and tiling brings around 2.6% overhead on average, the overall energy savings due to our approach becomes 0.82%, 8.11%, 0.13%, and 2.16% for the NU, ME, CO, and RE schemes, respectively. In other words, our approach is useful even if one considers all overheads and total energy consumption. Moreover, as will be shown shortly, these savings increase when we combine some of our optimizations.

Table IV. Normalized Energy Consumption of Different Scheme Combinations with Respect to OD Approach

Benchmark	$NU + ME$	$NU + RE$	$ME + CO$
adi	71.0%	85.2%	67.8%
apsi	59.6%	59.9%	64.6%
bmcm	56.2%	81.4%	57.4%
btrix	59.8%	72.2%	65.1%
mxm	65.5%	90.1%	66.8%
tomcatv	57.3%	76.9%	63.1%
wss	64.4%	79.3%	61.5%

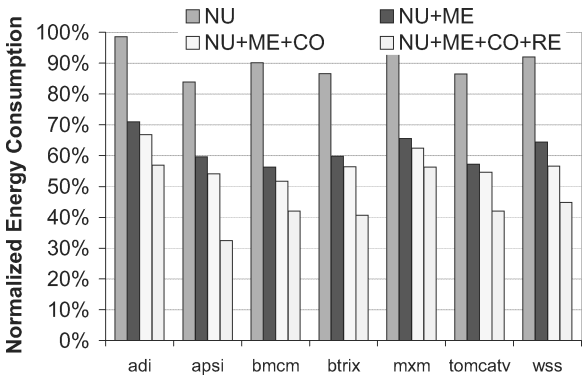

Fig. 13. Normalized energy consumption values with respect to the OD version.

Table IV shows the results with the different combinations of our schemes, namely, NU+ME, NU+RE, and ME+CO. One can observe from this table that when migration is allowed on a nonuniform banked memory (NU+ME), on average, it is possible to achieve a 38% reduction in energy (corresponding an average total energy savings of 10.39%). On the other hand, the combined ME+CO scheme is able to reduce the energy consumption by 36% on average (corresponding an average total energy savings of 9.71%). One observation from this result is that ME+CO generates better results than applying ME or CO individually, in all four applications; that is, data compression brings further benefits over data migration alone.

In the next set of experiments, we evaluate the effect of applying all the schemes in a cumulative manner. Figure 13 shows the normalized energy consumption values obtained using the NU, NU+ME, NU+ME+C0, and NU+ME+CO+RE schemes over the OD scheme. The average energy reduction under these schemes is 9.8%, 38.0%, 42.5%, and 55.0%, respectively. The last value corresponds to a 16.21% reduction of total energy consumption. As can be seen from this figure, although each scheme brings additional savings over the previous ones, there are marginal reduction decreases due to the limited amount of energy reduction potential.

Recall that the benchmarks we use in our experiments are extracted from scientific benchmarks suites. While the access patterns exhibited by these benchmarks are close to those exhibited by array-dominated embedded applications,

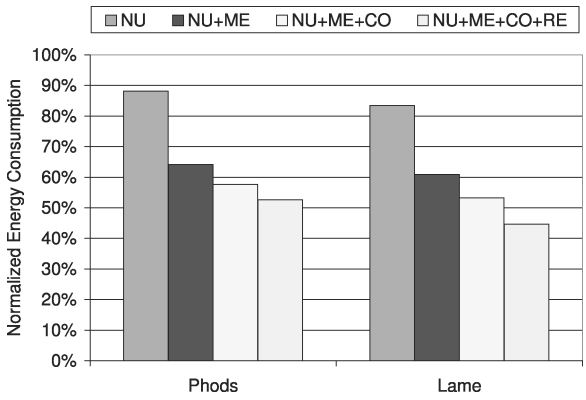

Fig. 14. Normalized energy consumption values with respect to the OD version.

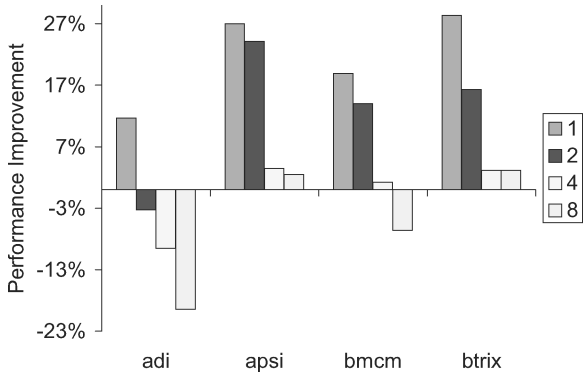

Fig. 15. Percentage improvements in performance brought by the NU- over the OD approach.

it is also important to check whether our technique can really bring benefits with embedded applications. To see this, we also conducted experiments with two full embedded applications. Phods is a hierarchical motion estimation implementation from Zervas et al. [1998]. Lame is, on the other hand, an MP3 encoder and is from the MiBench suite [MiBench 2001]. The normalized energy consumption results with these two benchmarks are presented in Figure 14 under the different optimization schemes. Our main observation is that the NU, NU+ME, NU+ME+C0, and NU+ME+CO+RE schemes save energy over the OD scheme for both benchmarks. These results are consistent with those presented earlier in Figure 13.

While the results presented so far indicate significant energy savings using our techniques, one may need to consider the performance aspect as well. Performance results are given in Figure 15. These values are given as improvements brought by the NU approach over the OD scheme, with different bank sizes. Negative values indicate a performance overhead if our approach is used. The average performance improvement values are −5%, 14%, 7%, and 13% for adi, apsi, bmcm, and btrix, respectively. One can observe from this figure that the maximum overhead brought by our approach is −19% for adi with a bank of

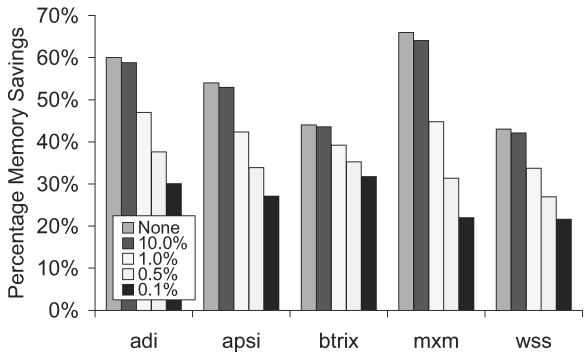

Fig. 16. Percentage memory-savings with different performance-overhead bounds.

size 8. On the other hand, the maximum improvement brought by our approach is 28% for btrix with a bank of size 1. In general, the smaller the bank size, the better the performance. This follows from the fact that the number of cycles to activate a bank (if the corresponding bank is in low-power operating mode) decreases if a smaller bank-size is employed. Also, the experiments with our two embedded benchmarks show that, on a two-bank system, the average performance improvements for Phods and Lame are 6.4% and 7.7%, respectively.

As stated earlier, another performance factor is the effectiveness of the decompression algorithm. In general, our approach minimizes the performance overhead due to decompressions by scheduling predecompressions earlier. However, this outcome depends on the latency of the particular decompression algorithm at-hand. If the predecompressions have to be scheduled very early, then the memory savings will reduce. To evaluate this fact we have implemented a version of the compress utility in UNIX, which is based on the LZC compression method. This is a specific implementation of LZW using variable-size pointers, as in LZ78. Figure 16 shows the memory savings using this technique based on a specific performance-overhead bound. We give the results with performance bounds ranging from no bound to 0.1%. As can be seen from this figure, without any performance limitation we can achieve 53% memory savings on average. Even employing a 10% performance overhead gives us 52% memory savings, on average. This performance overhead comes from the fact that we need to reemploy predecompression and the latencies of such predecompressions cannot be hidden all the time. The last bars on this graph show the memory savings when we have the performance-overhead bound as 0.1%. We achieve, on average, 26% memory savings using this bound. One key observation in this graph is the drastic reduction in the memory savings with mxm. With 10% performance-overhead we observe 64% memory savings, whereas these savings drop to 22% with 0.1% performance overhead. This follows from the fact that the data blocks used in mxm are larger when compared to other benchmarks (78KB versus 39.1KB). Hence, the compression ratios are higher due to larger data blocks, whereas the decompression latencies are longer. Higher decompression latencies result in performance limitations and reduce the memory savings, with tighter performance bounds.

Table V. Percentage Reduction in Energy of NU-over OD Approach with Varying (fixed) Bank Sizes

Benchmark	*Bank Size*				*Average*
	1	2	4	8	
bmcm	9.8%	12.9%	12.3%	14.1%	12.3%

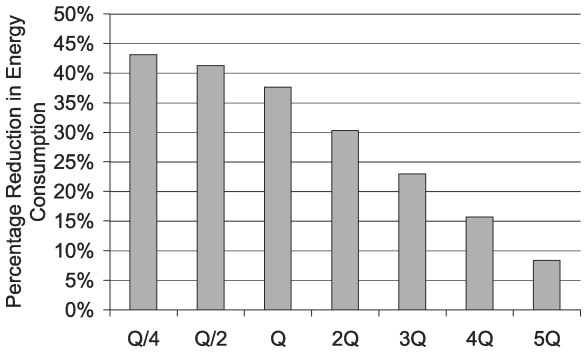

Fig. 17. Percentage improvement with different migration-cost ratios (bmcm).

7.3 Sensitivity Analysis

In this section, we modify some of the default simulation parameters (see Table III), and conduct a sensitivity analysis. Unless stated otherwise, we change only one parameter at-a-time; the remaining parameters are the same as in Table III. While, due to space concerns, we focus mostly only on the bmcm benchmark, our observations extend to the remaining benchmarks as well. Table V gives the energy reductions brought by the NU- over the OD version for different bank sizes. Bank sizes are given in terms of the data-block sizes shown in Table II. For example, in the third column (entitled "2"), we compare our approach with the OD strategy with banks of fixed size $2k$ (assuming that a data-block size is k). In NU, a total of $8k$ memory is partitioned nonuniformly. However, under the uniform-banking scheme, memory is composed of four different banks, where all banks are of size $2k$. The sixth column gives the average improvement brought by our approach over the OD scheme. We see that the overall average reduction in energy consumption is 12.3%, demonstrating the benefits brought by the NU scheme. As stated earlier, we also used ILP to obtain the results for the OD scheme in order to make a fair comparison against our approach.

The bar chart in Figure 17 shows the percentage improvement in energy achieved by ME over OD under the different cost ratios. On the x-axis of this graph, Q denotes the default energy cost for migration. We see from these results that the effectiveness of our approach increases as the relative cost of migration decreases. More specifically, the best energy savings are obtained with a migration cost of $Q/4$, whereas the worst savings occur with a migration cost of $5Q$, as expected. It should be observed that even with the relative migration cost of $5Q$, our approach achieves 8% reduction on energy consumption over the OD scheme, which implements optimal data placements across banks.

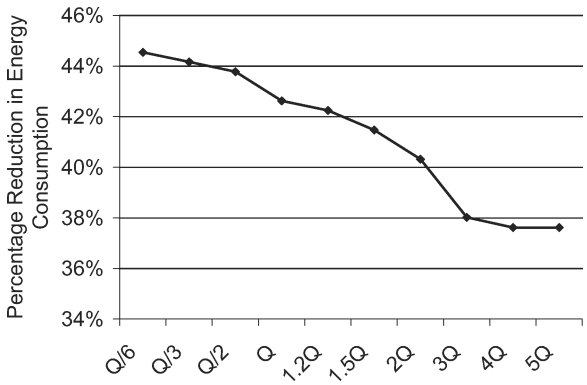

Fig. 18. Percentage improvement with different compression/decompression-cost ratios (`bmcm`).

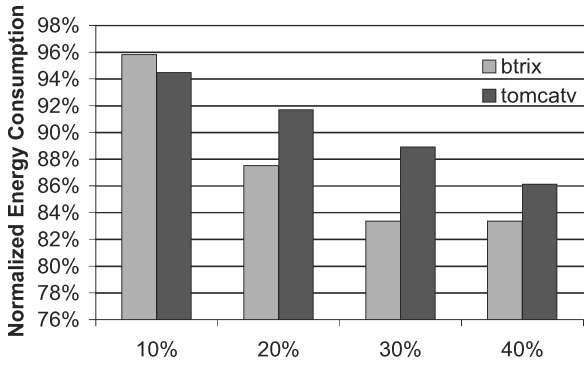

Fig. 19. Normalized energy consumption values with different replication limits, R_{lim} (`btrix` and `tomcatv`).

The graph in Figure 18 shows the percentage reduction in energy achieved by CO over OD under the different compression/decompression costs. Again, the value Q on the x-axis corresponds to the original compression/decompression cost assumed. We see from these results that as the relative cost of compression/decompression decreases, the effectiveness of our approach increases. The energy savings achieved with a cost of $Q/6$ is slightly less than 45%. On the other hand, energy savings with a cost of $5Q$ is about 38%, which is equal to the savings with the ME approach. One can observe from this graph that both costs $4Q$ and $5Q$ result in a memory-system energy savings of around 38%. This is due to the fact that compression/decompression is no longer used when the relative compression/decompression cost is set $4Q$, since in this case compression/decompression consumes much more energy than it can potentially save.

The bar chart in Figure 19 shows the normalized energy consumption of RE-over the OD version with different replication limits for btrix and tomcatv. Percentages on the x-axis of this graph denote R_{lim}, the maximum replication allowed, with respect to the total memory size. Recall that the default replication limit assumed in this article is 20% (see Table III). We see from these results that the effectiveness of our approach increases with the replication

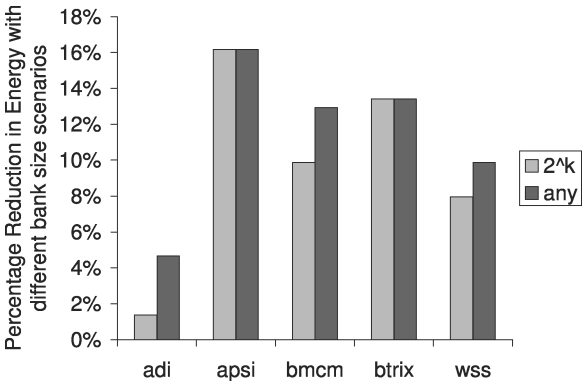

Fig. 20. Percentage reduction with different bank sizings.

limit. However, this also increases the memory-space consumption of the applications. Another problem with achieving higher percentages is that, beyond a certain percentage (depending on the application at-hand), replication does not provide much additional benefit. For example, in `btrix`, the energy savings increase by 0.1% when moving R_{lim} from 30% to 40%. Overall, these results show the tradeoff between memory consumption and energy savings when data replication is used. To summarize, our experiments clearly show that the proposed techniques save energy, both individually and when applied together.

Recall that all the results reported so far were obtained using memory-bank sizes that are powers of two. If we were to relax this assumption so that we could use all possible bank sizes (not only powers of two), we get the results shown in Figure 20. This graph shows the average percentage improvement obtained by the NU-over the OD version with banks with powers-of-two size (1,2,4,8) and with banks of any size (1,2,3,4,5,6,7,8). Note that the total memory size is kept constant and the only difference is in the way it is partitioned into the banks. One can observe from these results that the improvement brought by our approach over the OD version is better if we do not restrict the bank size to powers of two, due to the greater flexibility obtained. For example, if three data blocks are accessed frequently together, then it might be wise to keep them together in a bank which can hold all three. At the same time, it is better not to have any space left in the bank, which consumes extra energy. In this case, instead of using a bank size of 2 or 4, it is more profitable to use one of size 3.

As stated earlier, so far in our experiments we have used only the power-down mode. The graph in Figure 21 shows experimental results with the default parameters (Table III) for nap and standby modes, as well as for the power-down mode (the default), when only given type of mode and using higher-energy modes. Even with the standby mode, the *ME*- and $ME + CO$ approaches achieve 23% and 26% energy improvements, respectively. Although the power-down operating mode achieves the most energy savings (38% and 43%), it does not offer a great improvement over the nap mode (35% and 31%). This can be expected based on the bank energies given in Figure 2, and is due to the limited

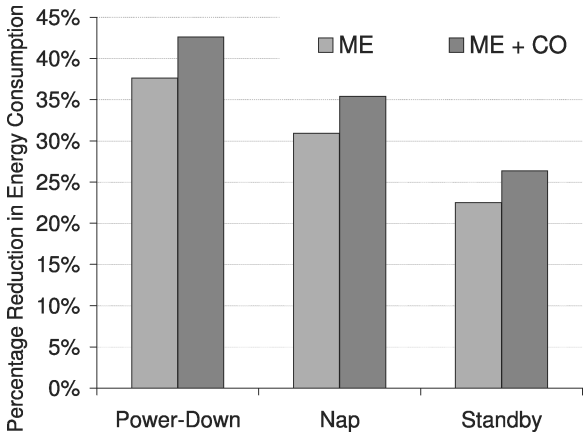

Fig. 21. Percentage improvement with different low-power modes (bmcm).

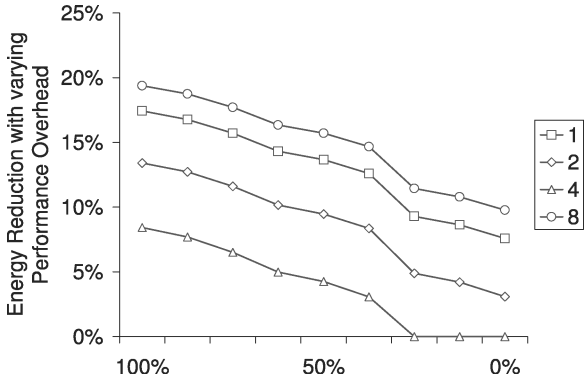

Fig. 22. Percentage reduction in energy with the different multiple low-power operating modes brought by the NU- over the OD approach.

number of banks that can be switched to the low-power operating mode. For example, with a two-bank memory, the maximum energy-savings possible is less than 50%, assuming that memory is always accessed (i.e., that at least one of the banks will be in active mode). In addition, bank reactivations and deactivations further decrease the potential energy-savings.

Instead of using a single low-power mode, as has been the case so far, we could use multiple power modes. In this case, each memory bank can be in one of the four modes listed in Figure 2 (active, standby, nap, or power-down) at any step. Figure 22 shows the energy reduction brought by our approach. This graph gives the energy improvement with respect to different performance constraints (x-axis) for different numbers of banks (1, 2, 4, 8). Specifically, the maximum performance-overhead allowed for the design is given on the x-axis. We observe that the energy savings vary between 0.0% to 19.4%. Note that if performance had not been a factor to consider, one could always have chosen the power-down mode (the most aggressive one), since it normally provides the least energy consumption.

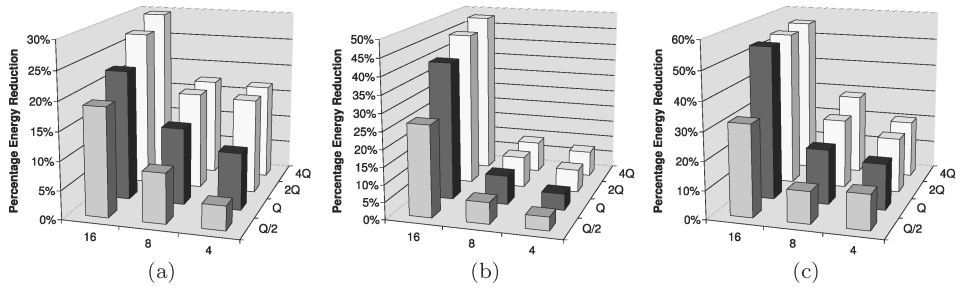

Fig. 23. Percentage energy reduction with differing memory capacities in terms of data-block-size ratios and number of steps for: (a) 2 banks; (b) 4 banks; and (c) 8 banks (btrix).

The final set of experiments study the impact of data-block- and step size on our results. The set of graphs in Figure 23 shows the results with different memory capacities; specifically, with data-block-size ratios and with a differing number of steps. Figures 23(a), (b), and (c) show the results with the ME + CO version for two, four, and eight banks, respectively. In these graphs, the x-axis shows memory capacity in terms of a data-block-size ratio ranging from 4 to 16, whereas the y-axis shows the number of steps, ranging from Q/2 to 4Q. Note that Q is the default number of steps in our experimental analysis. We observe from these results that energy consumption decreases with data-block size. This is due to the fact that with smaller-sized data blocks, the data-block accesses and operations can be performed in a higher granularity. All approaches discussed so far will benefit from such higher granularity. As a second dimension we study the effect of the number of steps in our approach for Q/2, Q, 2Q, and 4Q numbers of steps. Increasing the number of steps enables capturing the data-access pattern in a more detailed fashion; this in turn increases the flexibility of migration, compression, or decompression. Note that increasing the number of steps beyond a certain point may not yield a substantial improvement. For example, consider the graph given in Figure 23(a). Changing the number of steps from 2Q to 4Q does not yield a reduction of more than 2.2%.

Although, so far thus our approach has assumed that the compiler is given a data-block size and a step size as an input, this process can be automated by compiler as well. The pseudocode given in Figure 24 shows how a compiler can automatically calculate the data-block- and step sizes. Note that one would need the complexity limit of the ILP formulation. There are two parameters indicating the complexity of an ILP formulation, namely, the number of constraints (C_{lim}) and number of variables (V_{lim}). In order to obtain the number of steps, the compiler can check the number of loop nests that are computationally intense. For example, if there are two loops with N^3 iterations each, then it would be better to consider each of these loop nests as a single step, hence there is a total of three steps. This follows from the fact that loops with higher computational complexity will have the dominating number of accesses. These loop nests can be further divided, but this will increase the complexity of our ILP formulation.

```
C_lim — ILP constraint limit;
V_lim — ILP parameter limit;
t — application to be executed;
l — loop nest;
    procedure step-block(t, C_lim, V_lim) {
      obtain the loop nests and their complexities in t
      steps = 1
      for all l ∈ t such that l.complexity = t.max
        select l ∈ L_target
        steps = steps + l
      }
      blocksize = maximum array size
      while (C(blocksize, steps) ≤ C_lim) and (V(blocksize, steps) ≤ V_lim)
        blocksize = blocksize/2
      }
    }
```

Fig. 24. Pseudocode for automatically selecting block- and step size.

Increasing the granularity of block size will also result in a more complex ILP formulation. As a second part of the pseudocode given in Figure 24, the compiler can iteratively double the number of blocks until the ILP complexity limits are reached. This way, the ILP-solution time can be kept reasonable and a better result is obtained. It is possible to implement better compiler techniques to decide on block- and step size, but our goal here is to show that a compiler can easily identify reasonable ones.

Although increasing the granularity of data-block- and step size will reduce the energy consumption, this will also incur performance overheads. First, all data accesses, movements, compressions, and decompressions will need to be performed multiple times. Second, higher granularity will increase the number of parameters/constraints in the ILP formulation, which will affect the solution time exponentially.

8. CONCLUSION

This article shows how ILP (integer linear programming) can be used for formulating the problem of optimal data migration, data compression, and data replication in a banked memory architecture with low-power operating modes. In this work, an ILP solver is connected to an optimizing compiler. The optimizing compiler provides the data-access pattern information, that it extracts from the application source-code, to the ILP solver, which in turn determines the optimal data migration, data compression/decompression, and data replication patterns. This article also presents experimental evidence demonstrating the impact of the proposed approach in practice. The results indicate that the ILP-based techniques presented in this article are very effective in reducing memory energy consumption, and that our savings are consistent across a range of values of major simulation parameters.

REFERENCES

ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. 2003. Self-Tuning wireless network power management. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 176–189.

AZIZI, N., NAJM., F. N., AND MOSHOVOS, A. 2003. Low-Leakage asymmetric-cell sram. *IEEE Trans. Very Large Scale Integr. Syst. 11*, 4, 701–715.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 83–94.

BUNDA, J., FUSSELL, D., AND ATHAS, W. C. 1995. Energy-Efficient instruction set architecture for CMOS microprocessors. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, 298.

CAO, Y., TOMIYAMA, H., OKUMA, T., AND YASUURA, H. 2002. Data memory design considering effective bitwidth for low-energy embedded systems. In *Proceedings of the 15th International Symposium on System Synthesis*, 201–206.

CATTHOOR, F., DE GREEF, E., AND SUYTACK, S. 1998. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic, Norwell, MA.

CHANDRA, S. AND VAHDAT, A. 2002. Application-Specific network management for energy-aware streaming of popular multimedia formats. In *Proceedings of the General Track: USENIX Annual Technical Conference.* USENIX Association, Berkeley, CA, 329–342.

CHANG, N., CHOI, I., AND SHIM, H. 2004. DLS: Dynamic backlight luminance scaling of liquid crystal display. *IEEE Trans. Very Large Scale Integr. Syst. 12*, 8, 837–846.

CHOI, I., SHIM, H., AND CHANG, N. 2002. Low-Power color TFT LCD display for hand-held embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 112–117.

DELALUZ, V., KANDEMIR, M., AND SEZER, U., eds. 2003. *Improving Off-Chip Memory Energy Behavior in a Multi-Processor, Multi-Bank Environment.* Lecture Notes in Computer Science, vol. 2624. Springer.

DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2000. Energy-Oriented compiler optimizations for partitioned memory architectures. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 138–147.

DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., SIVASUBRAMANIAM, A., AND IRWIN, M. J. 2001. DRAM energy management using software and hardware directed power mode control. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, 159.

DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. N. 1995. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing (MLICS)*, USENIX Association, Berkeley, CA, 121–137.

EUN LEE, J., CHOI, K., AND DUTT, N. D. 2003. Energy-Efficient instruction set synthesis for application-specific processors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 330–333.

FAN, X., ELLIS, C., AND LEBECK, A. 2001. Memory controller policies for DRAM power management. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design.* 129–134.

FAN, X., ELLIS, C., AND LEBECK, A. 2002. Modeling of DRAM power control policies using deterministic and stochastic Petri nets. In *Proceedings of the Workshop on Power Aware Computer Systems (PACS).* Springer.

FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. M. 2000. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 252–263.

FARRAHI, A., TELLEZ, G., AND SARRAFZADEH, M. 1998. Exploiting sleep mode for memory partitions and other applications. In *Proceedings of the VLSI Design Conference*, 271—287.

FLAUTNER, K., KIM, N. S., MARTIN, S., BLAAUW, D., AND MUDGE, T. 2002. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 148–157.

GHOSE, K. AND KAMBLE, M. B. 1999. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 70–75.

GREENAWALT, P. M. 1994. Modeling power management for hard disks. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS)*, 62–66.

GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. 2003. Reducing disk power consumption in servers with DRPM. *Comput. 36*, 12, 59–66.

HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. 1996. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking (MobiCom)*, 130–142.

INOUE, K., ISHIHARA, T., AND MURAKAMI, K. 1999. Way-Predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 273–275.

INOUE, K., MOSHNYAGA, V. G., AND MURAKAMI, K. 2002. A history-based i-cache for low-energy multimedia applications. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 148–153.

JEJURIKAR, R., PEREIRA, C., AND GUPTA, R. 2004. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, 275–280.

KAMBLE, M. B. AND GHOSE, K. 1997. Analytical energy dissipation models for low-power caches. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPED)*, 143–148.

KANDEMIR, M., KOLCU, I., AND KADAYIF, I. 2002. Influence of loop optimizations on energy consumption of multi-bank memory systems. In *Proceedings of the 11th International Conference on Compiler Construction*, 276–292.

KANDEMIR, M., RAMANUJAM, J., AND CHOUDHARY, A. 1999. Improving cache locality by a combination of loop and data transformations. *IEEE Trans. Comput. 48*, 2, 159–167.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*.

KESSELMAN, A., KOWALSKI, D., AND SEGAL, M. 2005. Energy efficient communication in ad hoc networks from user's and designer's perspective. *SIGMOBILE Mob. Comput. Commun. Rev. 9,* 1, 15–26.

KIM, S., VIJAYKRISHNAN, N., KANDEMIR, M., SIVASUBRAMANIAM, A., IRWIN, M. J., AND GEETHANJALI, E. 2001. Power-Aware partitioned cache architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 64–67.

KIM, W., KIM, J., AND MIN, S. L. 2004. Preemption-Aware dynamic voltage scaling in hard real-time systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 393–398.

LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. 2000. Power aware page allocation. *SIGPLAN Not. 35,* 11, 105–116.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HLLBERG, G., HGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *IEEE Comput. 35,* 2, 50–58.

MIBENCH. 2001. MiBench version 1.0. http://www.eecs.umich.edu/mibench/.

MOON, J.-S., ATHAS, W. C., BEEREL, P. A., AND DRAPER, J. T., eds. 2002. *Low-Power Sequential Access Memory Design.*

NEMHAUSER, G. L. AND WOLSEY, L. A. 1988. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York.

OZTURK, O., CHEN, G., AND KANDEMIR, M. 2006. Compiler-Guided data compression for reducing memory consumption of embedded applications. In *Proceedings of the Conference on Asia South Pacific Design Automation.*

OZTURK, O. AND KANDEMIR, M. 2005a. Integer linear programming based energy optimization for banked DRAMs. In *Proceedings of the ACM Great Lakes Symposium on VLSI*, 92–95.

OZTURK, O. AND KANDEMIR, M. 2005b. Nonuniform banking for reducing memory energy consumption. In *Proceedings of the Design Automation and Test in Europe (DATE)*, 814–819.

PANDA, P. R. 1999. Memory bank customization and assignment in behavioral synthesis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 477–481.

RAMBUS. 1999. 128/144-mbit direct rdram data sheet.

SAGHIR, M. A. R., CHOW, P., AND LEE, C. G. 1996. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 234–243.

SU, C.-L. AND DESPAIN, A. M. 1995. Cache design trade-offs for power and performance optimization: A case study. In *Proceedings of the International Symposium on Low Power Design (ISPLED)*, 63–68.

SUDARSANAM, A. AND MALIK, S. 2000. Simultaneous reference allocation in code generation for dual data memory bank ASIPS. *ACM Trans. Des. Autom. Electron. Syst. 5,* 2, 242–264.

WILSON, R. P., FRENCH, R. S., WILSON, C. S., AMARASINGHE, S. P., ANDERSON, J. M., TJIANG, S. W. K., LIAO, S.-W., TSENG, C.-W., HALL, M. W., LAM, M. S., AND HENNESSY, J. L. 1994. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not. 29,* 12, 31–37.

XPRESS. 2002. Xpress-mp. http://www.dashoptimization.com/pdf/Mosel1.pdf.

YE, W., VIJAYKRISHNAN, N., KANDEMIR, M., AND IRWIN, M. J. 2000. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Conference on Design Automation (DAC)*, 340–345.

ZERVAS, N. D., MASSELOS, K., AND GOUTIS, C. 1998. Code transformations for embedded multimedia applications: Impact on power and performance. In *Proceedings of the ISCA Power-Driven Microarchitecture Workshop.*

ZHUO, J. AND CHAKRABARTI, C. 2005. System-Level energy-efficient dynamic task scheduling. In *Proceedings of the 42nd Annual Conference on Design Automation (DAC)*, 628–631.

Received February 2006; revised September 2006; accepted November 2007